Incorporating Feedback Mechanism to Bayesian Networks-based Test Case Prioritization

By

Siavash Mirarab and Ladan Tahvildari
Software Technologies Applied Research (STAR) Group
Department of Electrical and Computer Engineering
University of Waterloo,
Waterloo, Ontario

TECH. REPORT UW-ECE#2007-27

Sept. 2007

Table of Contents

List of Tables

List of Figures

Abstract

An important aspect of regression testing is to prioritize the test cases for execution. This paper presents a new technique for prioritizing test cases in order to enhance the rate of fault detection. This new technique enhances our previously introduced approach based on Bayesian Networks (BN) [15]. The advantage of the new technique is in using a feedback mechanism to augment the model after each test execution. As a proof of concept, the proposed technique is applied to eight consecutive versions of a large-size software system. The obtained results indicate a strong increase in the rate of fault detection.

1 Introduction

Prioritizing existing test cases from earlier versions of a software system is one of the main techniques used to address the problem of regression testing. Such a technique uses the test-suite developed for an earlier version of a software system to conform the new added requirements in the current version. Selecting all or a portion of the test-suite to execute, so called Regression Selection Techniques (RST), can be very costly. By using RST, testers do not have the option to adjust their test-effort to their budget. To provide the necessary flexibility, researchers have introduced prioritization techniques [17, 21] by which testers can order the test cases based on certain criteria, and then run them in the specified order. During the past ten years, researchers have introduced many techniques for prioritization [9, 11, 13, 18, 19, 20].

Despite all the above-mentioned research, empirical studies indicate that there is a significant gap between optimal solutions to prioritization problem and proposed techniques [11]. One can imply from the past research that using more sources of information results in better performance. To fill this gap, we previously proposed a new probabilistic approach which utilizes Bayesian Networks (BN) [16] to incorporate three sources of information into one unified model: i) source code modification information, ii) univariate measures of fault-proneness, and iii) test coverage data [15]. We compared our technique to other common techniques from literature and found that when there are reasonable number of faults (more than one in that study) in the source code, our proposed technique achieves better values of APFD (Average Percentage of Faults Detected) [17].

However, the proposed technique did not employ a feedback mechanism [11], as opposed to other high-performance techniques. In this report, we present a novel test-case prioritization technique which is based on our previous approach, but makes use of feedback. Here, feedback means that after adding each test case to the order, other test-cases which cover the same elements as the added test-case get less chance of selection.

The rest of this article is organized as follows. Section 2 introduces test case prioritization in more detail. In Section 3, our proposed approach to solve the prioritization problem is presented. Section 4 discusses the obtained results after applying our techniques on our case study. Finally, we make our conclusions and point out some future directions for this research work.

2 Test Case Prioritization

The classic definition of test case prioritization is based on finding a permutation of test cases which can maximize an award function reflecting the goal of prioritization [17]. Kim *et al.* look at the same problem from a probabilistic point of view. Adopting the probabilistic nature of their description, a prioritization can be considered as:

1. Gathering "useful" evidences $\mathcal{E}_i$s from software system
2. Using a "prioritization technique" to assign a probability of success to each test case T_i in test-suite $\mathbb{T}$, given all the evidences $P(t_i|\mathcal{E}_1, \ldots, \mathcal{E}_n)$
3. Selecting and running test cases from $\mathbb{T}$ based on the defined probability model

"Useful" evidence in step 1 refers to all information the "prioritization technique" of step 2 is interested in (e.g. test coverage information). In step 2, the main part of this process, we have a set of random variables $\mathbf{t_i}$, each of which reflect the outcome of a test case T_i from $\mathbb{T}$. These variables have two possible values: "Success" (meaning that a defect is detected) and "Failure". The event of "Success" in T_i is denoted as t_i. A "prioritization technique" is a systematic way to estimate all $P(t_i|\mathcal{E}_1, \ldots, \mathcal{E}_n)$ values. For example, in coverage-based techniques, the number of elements covered by any test case is used to estimate the probability of success for that test case:

$$P(t_i|\mathcal{E}) = \frac{the\ number\ of\ elements\ covered\ by\ \mathrm{t}_i}{total\ number\ of\ elements}$$

In step 3, the estimated probabilities are used to select test cases for execution. The simplest way to do so would be ordering test cases according to the probabilities and then starting from the beginning of the ordered list. The other way is to choose test cases randomly but based on the estimated probability distribution. This view of prioritization is a simplified version of the one presented in [15]. Our approach in prioritizing test cases is based on this view of the prioritization problem and hence makes use of probabilistic tools.

2.1 Feedback

Many of the prioritization techniques make use of a "feedback" mechanism. These techniques, often called "additional" techniques, have a greedy and iterative approach where after adding each test case to the order they estimate the effects of this selection for further selections. Conceptually, feedback means updating our beliefs about the system after adding each test case to the prioritized order.

For example, lets assume we have a system with six elements: $e_1 \ldots e_6$. Consider the coverage relation between test cases and elements is as follows: $t_1 \rightarrow \{e_2, e_5\}, t_2 \rightarrow \{e_1, e_3\}, t_3 \rightarrow \{e_4, e_5, e_6\}$. If our prioritization is merely based on coverage, the first chosen test case would be t_3 because it covers three elements while the others cover two. Then, we have two test cases left both of which cover two elements and hence we should select randomly between them. However, we know that e_5 is already covered by t_3; therefore we can say t_1 has merely one *additional* coverage whereas t_2 has two. Therefore it is wiser to first choose t_2 and then go to t_1. This notion of using *additional* coverage illustrates what feedback mechanism means. After adding t_3, we can update our beliefs about the coverage power of each test case such that already tested elements do not effect later selections.

3 Proposed Approach

Figure 1 illustrates a high-level schema of our approach for test case prioritization. The first step, *Extracting Evidences*, gathers all useful information that needs to be included in the model. Our current solution exploits three sources of information: *software quality metrics*, *test coverage measures*, and *change analysis data*. In the next step, *Building Bayesian Network*, we build an inclusive probabilistic model to relate the data. The details of how the model is built will be elaborated further.

The third step, *Ordering Test Cases*, is where feedback mechanism can be incorporated to the technique. This step consists of two tasks. The first task, *Probabilistic Inference*, employs the probabilistic inference algorithms to associate to each test case its probability of success given the collected evidences. On the non-additive version of our approach, here the process terminates and the final order

is determined using the extracted probabilities. On additional version, however, after the inference, the test case with the highest probability of the success is selected and added to the final order. Then, we go to *Updating Bayesian Network* step, where the BN model gets updated based on the test case just added to the final order. This update should be such that other test cases which cover similar parts of the code get less probability of selection. The details of how this goal is archived will be described in section 3.2.2. Then, we go back to the *Probabilistic Inference* step and a new iteration begins. These iterations continue until all test cases are added to the final order. This loop is what brings the feedback mechanism to our approach.

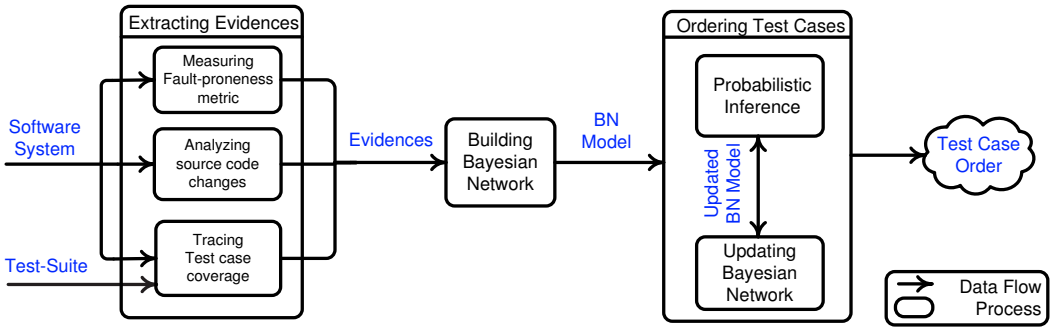

Figure 1. Steps of the Proposed Approach.

Note that the first step of our approach is already well established, and we will just make use of the existing contributions to implement it. In the rest of this section, we elaborate on the second phase of our proposed process (with a short introduction to Bayesian Networks) and then explain the third step through which the feedback mechanism is introduced.

3.1 Building Bayesian Network

A Bayesian Network (BN) [16] is a directed acyclic graph consisting of three elements: *nodes* representing random variables, *arcs* representing probabilistic dependency among those variables, and a *Conditional Probability Distribution Table (CPT)* for each variable, given its parents. The nodes can be either evidence or latent variables. An evidence variable is a variable where in we know its values (i.e. it is measured). Arcs specify the causal relation between variables. Each node has a table called CPT which includes the probabilities of outcomes of its variable given the values of its parents.

Bayesian Networks reflect the expert belief about the problem domain. They can be used to answer probabilistic queries. For example, based on the evidence (observed) variables, the posterior probability distributions of some other variables can be computed (probabilistic inference). There are two facets to building a BN, *designing the structure* and *computing the parameters*. Regarding the first issue, the notions of conditional independence and causal relation can be of great help. Intuitively, two events (variables) are conditionally independent if knowing the value of some other variables makes the outcomes of those events independent. For computing the parameters, expert knowledge, statistical learning, and probabilistic estimations can be used. One potential problem is that we may know how a variable is dependent on each of its parents, but we do not have its distribution conditioned on all parents. In these situations, the “noisy-OR” assumption can be helpful. The “noisy-OR” assumption gives the

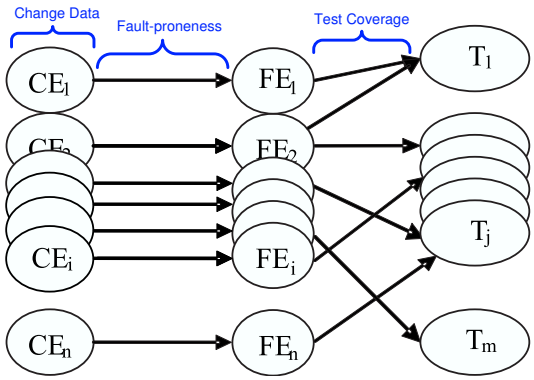

Figure 2. The Structure of the BN.

interaction between the parents and the child a causal interpretation and assumes that all causes (parents) are independent of each other in terms of their influence on the child [16].

Modeling is the main focus in solving the problems using BN. A description of how three basic elements of a BN (nodes, arcs, and CPTs) are designed in our approach is as following:

3.1.1 Nodes

There are three categories of nodes in our models:

• $\mathbf{ce}$: These variables represent change in the elements of the program. Each software element in the considered level of granularity (i.e. a class) has a node of this type. These variables can take a value of "Changed" (denoted to by ce_i) or "Unchanged"(denoted to by $\neg ce_i$).

• $\mathbf{fe}$: This category reflects our belief whether each element is faulty. Each element of the program has one node of this type and each node can have the values of "Faulty" (fe_i), or "Non-Faulty"($\neg fe_i$).

• $\mathbf{t}$: These variables represent the outcome of a test case which can be "Success"(t_i) or "Failure"($\neg t_i$). Each test case has one node of this type.

3.1.2 Arcs

Each arc in a BN indicates a causal relation between variables of two connected nodes. There are two set of arcs in our network:

• $\mathbf{ce - fe}$: Each $\mathbf{fe}$ node is the child of the corresponding (i.e. of the same code element) $\mathbf{ce}$ node. The existence of these arcs reflect the causal relation that changes to elements of software can introduce faults in the same element.

• $\mathbf{fe - t}$: Each $\mathbf{t}$ node can be child of some $\mathbf{fe}$ nodes. These arcs imply the causal relation between presence of fault in a software element and success the of test cases that examine that element.

In Figure 2, the overall structure of the designed model is illustrated. Each $\mathbf{ce}$ node is connected to one $\mathbf{fe}$ node and the $\mathbf{fe}$ nodes are connected to an arbitrary number of $\mathbf{t}$ nodes.

3.1.3 Conditional Probability Table

Each node type has its own Conditional Probability Table (CPT):

• $P(\mathbf{ce}_i)$: $\mathbf{ce}$ nodes are not the child of any other node, so their distribution is not conditional. In our model, $\mathbf{ce}_i$ variables mean the effective change of the element.

$$P(ce_i) = ChangeIntensity(e_i)$$

In this formula, $ChangeIntensity(e_i)$ is a function which returns how much semantic change the element e_i has gone through. This function can be implemented with algorithms as simple as Unix *diff* command. In our study, we have used an algorithm presented in [7] which uses byte code to estimate similarity between two versions of a program.

• $P(\mathbf{fe}_i|\mathbf{ce}_i)$: Considering that both $\mathbf{fe}$ and $\mathbf{ce}$ can take two values, the CPT will contain 4 values, two of which are trivial, since $P(fe_i|\mathbf{ce_i}) = 1 - P(\neg fe_i|\mathbf{ce_i})$. Therefore, we need to estimate two values: $P(fe_i|ce_i)$ and $P(fe_i|\neg ce_i)$. In general, the probability of presence of fault in software is called fault-proneness. It is empirically shown that one can approximately predict the fault-proneness of code elements using software metrics [5]. The aforementioned studies (and also an empirical study on the relation between APFD and software metrics [10]) indicate that measures of complexity and coupling are better indicators of fault-proneness. One specific study [12] has shown that coupling is a significantly better measure than other metrics. Here, we use measures of coupling as an indicator of fault-proneness:

$$P(fe_i|ce_i) = \frac{\alpha\, CBO(e_i)}{\max(CBO(e_x))} + \delta_1,\ (\alpha + \delta_1 \leq 1)$$

In this formula, CBO (Coupling between Objects) is an object-oriented metric from Chidamber and Kemerer suite [6] which counts the number of classes to which a given class is coupled (i.e. uses its methods and/or fields). The choice of this metric is based on the above-mentioned empirical studies [12]. The dominator is a normalization factor and α and δ_1 bound the probability of fault introduction.

As for $P(fe_i|\neg ce_i)$, estimating this value is harder because it represents the less probable situation that an element is faulty, even though it is not changed. In our modelling, we use the following formula:

$$P(fe_i|\neg ce_i) = \frac{\beta\, DIT(e_i)}{\max(DIT(e_x))} + \delta_2\ ,\ (\beta + \delta_2 \ll \alpha + \delta_1)$$

Here, DIT (Depth of Inheritance Tree) is another metric from Chidamber and Kemerer suite [6] which measures the maximum length of a path from each class to a root class in the inheritance structure. We use this metric in order to capture change impacts from super classes to child classes. The important invariant is that the probability of fault presence in unchanged elements should be much less than in changed elements. Let $\gamma = \frac{\alpha+\delta 1}{\beta+\delta 2}$. By adjusting γ (the change effect factor) we can control the degree to which the presence of change in an element raises our belief in its fault-proneness.

• $P(\mathbf{t}_i|\mathbf{fe}_1 \ldots \mathbf{fe}_n)$: $\mathbf{t}$ nodes can have more than one parent because a test case may be able to find faults from different elements of the software. The probability of fault detection for each test case can be estimated using its coverage of faulty elements. Having test coverage values, we estimate:

$$P(t_i|fe_j) = Cov(t_i, e_j)$$

where $Cov(t_i, e_j)$ is a function returning the percentage of the code element e_j covered by test case T_i. This formula estimates the relation between a test case and one single element. However, to build the CPT we need the probability of success for a test given all combinations of values of its parent $\mathbf{fe}$ variables. In order to cope with this problem, we make the noisy-OR assumption, explained above. In simple terms, the assumption is that the relation of a test case to an element is independent from its relation to any other element. It can be argued that the ability of a test case to reveal a fault in one element is not related to its fault revealing ability in other elements, hence the assumption. Having the noisy-OR assumption, we can find all $P(t_i|\mathbf{fe}_1 \dots \mathbf{fe}_n)$ values. The detailed description of how the noisy-Or assumption works can be found in [16].

3.2 Ordering Test Cases

Once the BN models are built, we need to use them to order test cases. This is done iteratively, where in each iteration:

1. Using probabilistic inference the probability distribution of $\mathbf{t_i}$ variables are found.
2. a certain number (stp) of test cases with the highest probabilities are selected and added to the prioritized order.
3. the model is updated based on the added test cases (feedback).

In Figure 1, we saw two steps which correspond to first and third tasks in this loop. In the following, we first elaborate more on the two mentioned steps and then describe some interesting facts about the second item and how the stp parameter can be used.

3.2.1 Probabilistic Inference

In this step we estimate the $P(t_i)$ values using the probabilistic inference described in section 3.1. A BBN can be used to answer probabilistic queries. That is, given some new knowledge for some observed variables (evidence variables), we can request the updated probability distribution for other variables. This process is called probabilistic inference and many algorithms have been developed in BN literature to perform it [16]. These algorithms fall into two categories: exact and sampling algorithms. While exact algorithms are guaranteed to find the exact probabilities, the sampling algorithms estimate those values through iterative sampling of data. Depending on the structure of the network and also how many times the sampling is performed, the sampling algorithms can provide less execution time.

3.2.2 Updating Bayesian Network

This step is introduced to add feedback to the approach. After each round of inference the element that has the most probability of success would be added to the order. Next, in *Updating Bayesian Network* step, the variable corresponding to that node would be marked as an evidence node. The value of this evidence node would be "Failure". This way, the probability of existence of fault in elements covered by that test will be reduced and the other test cases that test the same elements will see a decrease in the probability of success. Note that regardless of whether a test case succeeds or not, we mark it as a "Failure". There are two reasons for this: first, during prioritization we do not know yet whether the test

case is going to succeed. Second if we mark a node as "Success", then the other test cases that cover the same elements will have a greater chance of selection. This is not desirable because we already know that a bug exists in those elements.

3.2.3 stp Parameter

Task 2 from from the aforementioned loop is a trivial task except for an interesting parameter that it introduces: stp. This parameter controls how often the inference step is performed while adding the test cases to the order. On one head of the spectrum if wet the parameter to the size of the test suit, the technique gets reduced to non-additional and no feedback would be used. On the other end when we set the parameter to 1, feedback is used each and every selection. This parameter gives the practitioners the flexibility to adjust the running time of the technique itself. If they need the prioritization task to be finished very fast they can set the parameter high, otherwise they can use lower values for stp.

By adjusting parameters such as stp, the described approach can be used to develop many alternative BN-based prioritization techniques. Here we aim at evaluating the feedback mechanism and hence consider two different options: first, *BN* which does not use feedback at all (i.e. $stp = |testsuit|$). Second, BN_A (A for additional) which utilizes feedback mechanism. The performance of these two techniques are the subject of our experimental evaluation.

4 Experimental Evaluation

To evaluate the proposed approach, we built a semi-automated environment for test case prioritization. As a proof of concept, eight consecutive versions of Apache Ant [1] with a catalogue of several prioritization techniques were examined.

4.1 Experiment Setup

• **Subject Program**: As our subject program, we used Apache Ant from "Software-artifact Infrastructure Repository (SIR)" built by Do *et al.* [8]. Like all other Java programs in SIR, Apache Ant has hand-seeded faults. We choose this program for our study because it has the largest number of faults and also is of a reasonable size (Table 1).

Table 1. Statistics on Apache Ant Case Study.

Metric Name	v0	v1	v2	v3	v4	v5	v6	v7
Faults Count	0	1	1	2	4	4	1	6
Test-suit Size	0	28	34	52	52	101	104	105
LOC (K)	23	37	57	57	95	97	97	124

• **Prioritization Techniques**: In this study, nine prioritization techniques listed in Table 2 are examined. The first three of them are control techniques: Optimal is the best possible order computed in

a greedy manner; Random orders randomly (the average of 50 runs); and Original is the original order of test cases. The next four techniques are conventional techniques based on [9] and all use coverage information. Two of them make use of change information (in addition). All of the techniques use class level coverage information. These techniques can use method and basic block level information as well, but previous experiments show this dimension does not affect the results significantly [15].

BN technique represents non-additive BN-based approach where the parameters of the technique are set as: $\alpha = 0.8$, $\delta_1 = 0.1$, $\beta = 0.1$ and $\gamma = 8$. Finally, BN_A is the new proposed technique where stp parameter is set to 1. In all cases exact algorithms are being used.

Table 2. Prioritization Techniques

Name	Evidences	Feedback
Optimal	Fault Matrix	Yes
Random	Nothing	No
Original	Nothing	No
C_Cov	Coverage	No
C_A_Cov	Coverage	Yes
Chg_Cov	Coverage+Change	No
Chg_A_Cov	Coverage+Change	Yes
BN	Coverage+Change+Fault-proneness	No
BN_A	Coverage+Change+Fault-proneness	Yes

• **Used/Developed Tools**: To implement our approach, a semi automated framework is built. In some steps, we used available tools. To collect software metrics, ckjm [3] is used; for gathering coverage information Emma [4] is utilized and the change information is obtained from Sandmark [7]. Also, we built and manipulated Bayesian Networks using Smile Library [2]. The rest of the approach was developed by us.

• **Measurement Criteria**: To be able to compare our results to other empirical studies (esp. those of [9]), APFD is used as the evaluation metric. This metric aims at calculating the fault detection rate by measuring the weighted average of the percentage of faults detected over the test-suite execution period. Its precise definition can be found in [17]. However, this metric has some drawbacks; for example, it neither takes into account the cost of each individual test nor the severity of faults.

4.2 Discussion on Obtained Results

The results of the case study are depicted in Figure 3 as a Boxplot diagram. In this diagram, for each of the techniques, we can see the distribution of gained APFD measures computed over the eight consecutive versions of Apache Ant. Also, Table 3 shows the average and standard deviation of APFD values over different versions for each technique.

From the Table 3 and Figure 3, it is evident that all techniques perform better than the “random” and “original” (the control techniques). It is also observable that techniques that use change information do not produce a significantly better result. On the other hand, those that employ the feedback mechanism (or “additional techniques”) bring about much better results than other techniques. They result in around 20% improvement on average compared to corresponding techniques. More interestingly, BN is the best

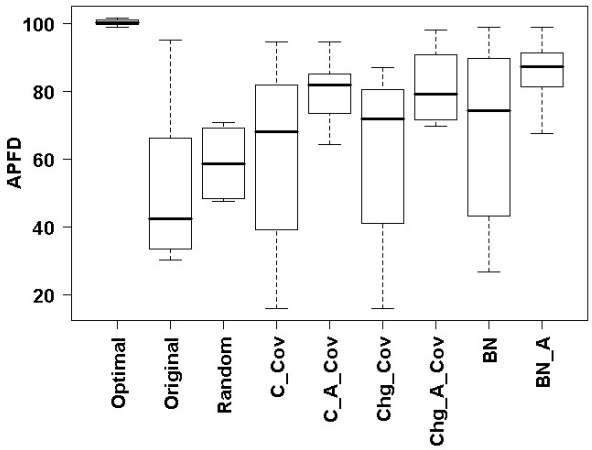

Figure 3. Boxplot Diagram of the Results

technique among all non-additional techniques while BN_A is the best among additionals. These two proposed techniques outperform the other techniques of their family by almost 5% on average (Table 3) and almost same value in median (Figure 3). In particular, the BN_A technique archives better average and median APFD values than any other technique. This technique has shortened the gap between the optimal solution and the practical techniques.

Table 3. Comparison among Used Techniques

Technique	Average	Standard Deviation
Optimal	100.48	1.03
Original	52.63	24.44
Random	58.97	10.78
C_Cov	60.19	30.76
C_A_Cov	79.80	10.56
Chg_Cov	59.80	29.16
Chg_A_Cov	81.85	11.74
BN	66.68	29.40
BN_A	85.70	10.05

We believe the reason why BN and BN_A techniques are performing better than others is that they use three sources of information. Furthermore, they model the causal relation between the used information as conditional probabilities. However, the fact that these two models use many sources of information leads to higher costs because the main cost of prioritization process is associated with the information extraction part. Therefore, the decision on which technique to use calls for a cost-benefit analysis. Models of cost-benefit analysis for prioritization techniques have been already proposed [9, 14].

Another interesting aspect of the obtained results is the effect of the number of faults in the system on the performance of techniques. We depicted APFD values of non-control techniques versus the number of faults in Figure 4. This figure suggests when the number of faults in the system grows, the APFD value of “additional” techniques decreases; whereas the others see an increase in their value of APFD.

This suggests that the feedback employing techniques perform better when smaller numbers of faults are available, but as the potential number of faults grows, non-additional techniques are more promising. This finding should be further inspected empirically on the systems that contain a larger variety of fault counts. If approved, this phenomenon can have a very important practical implication. If the software system is expected to contain many bugs, then non-additional techniques are of more use. On the other hand, so called "additional" techniques are more appropriate for the more steady stages of software development where developers struggle to find the last residual faults.

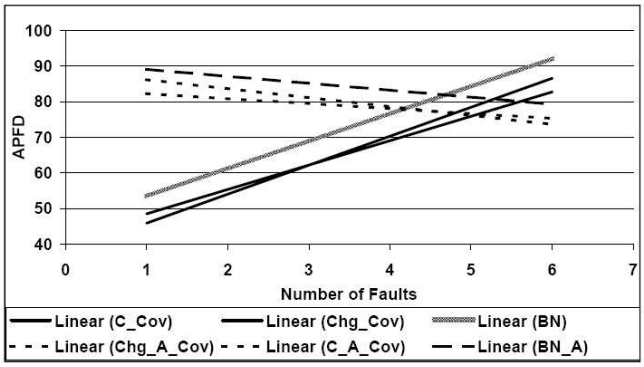

Figure 4. APFD versus Fault Counts

In conclusion, the two techniques that are based on our proposed approach are the best in their family of techniques. BN technique is performing better than all non-additional techniques. Also, BN_A, the proposed technique of this paper, resulted in the best prioritization according to APFD metric.

5 Conclusions and Future work

In this paper, we introduced a new technique for test case prioritization. This new technique adds a feedback mechanism to our previously introduced approach utilizing Bayesian Networks to solve the test case prioritization problem. We described our modelling approach in detail and introduced a framework to implement the approach. We performed a case study on eight versions of a large-size Java system and presented its results. The results suggest that the two techniques that are based on our proposed approach can produce the highest values of APFD in their category of techniques (two categories of additional and non-additional techniques). BN_A, the new technique introduced in this paper, prioritizes test cases better than any other technique as evaluated by APFD metric.

In future research, first the results should be further inspected using empirical experiments and taking into account cost-benefit models. Also, the software faults in this case study are all hand-seeded and their representativeness of real faults may be argued, therefore it is critical to evaluate this approach on programs that contain a reasonable number of real faults. The other way to extend this work is to use metrics for fault-proneness and change analysis and compare their performance with the presented ones. Finally, the impact of different parameters in the approach such as stp should be further inspected.

The presented work indicates using more mathematical approaches to model regression testing can result in significant improvements.

References

[1] Apache Ant, 2005. http://ant.apache.org.

[2] Genie/Smile, 2005-2006. http://genie.sis.pitt.edu/.

[3] CKJM, 2006. http://www.spinellis.gr/sw/ckjm/.

[4] Emma, 2006. http://emma.sourceforge.net/.

[5] L. Briand and J. Wüst. Empirical studies of quality models in object-oriented systems. *Advances in Computers*, 56:98–167, 2002.

[6] S. R. Chidamber and C. F. Kemerer. Towards a metrics suite for object oriented design. In *Proceedings of the Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 197–211, 1991.

[7] M. S. Christian Collberg, Ginger Myles. An empirical study of java bytecode programs. Technical Report TR04-11, Department of Computer Science, Univeristy of Arizona, 2004.

[8] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[9] H. Do, G. Rothermel, and A. Kinneer. Prioritizing JUnit test cases: An empirical assessment and cost-benefits analysis. *Empirical Software Engineering: An International Journal*, 11(1):33–70, 2006.

[10] S. Elbaum, D. Gable, and G. Rothermel. Understanding and measuring the sources of variation in the prioritization of regression test suites. In *Proceedings of the IEEE International Symposium on Software Metrics (METRICS)*, pages 169–179, 2001.

[11] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, 2002.

[12] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, 2005.

[13] J.-M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the ACM International Conference on Software Engineering (ICSE)*, pages 119–129, 2002.

[14] A. G. Malishevsky, G. Rothermel, and S. Elbaum. Modeling the cost-benefits tradeoffs for regression testing techniques. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 204–213, 2002.

[15] S. Mirarab and L. Tahvildari. A prioritization approach for software test cases based on bayesian networks. In *Fundamental Approaches to Software Engineering (FASE)*, LNCS 4422, pages 276–290. Springer-Verlag, 2007.

[16] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[17] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, 2001.

[18] D. Saff and M. D. Ernst. Reducing wasted development time via continuous testing. In *Proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 281–292, 2003.

[19] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 97–106, 2002.

[20] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos. Timeaware test suite prioritization. In *Proceedings of the IEEE International Symposium on Software Testing and Analysis (ISSTA)*, pages 1–12, 2006.

[21] W. E. Wong, J. R. Horgan, S. London, and H. A. Bellcore. A study of effective regression testing in practice. In *Proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 264–274, 1997.